Elon's Toys

Welcome to Elon's Toys on Exercism's Go Track. If you need help running the tests or submitting your code, check out `HELP.md`. If you get stuck on the exercise, check out `HINTS.md`, but try and solve it without using those first :)

Introduction

A method is a function with a special *receiver* argument. The receiver appears in its own argument list between `func` keyword and the name of the method.

```
func (receiver type) MethodName(parameters) (returnTypes) {
}
```

You can only define a method with a receiver whose type is defined in the same package as the method.

```
package person

type Person struct {
        Name string
}

func (p Person) Greetings() string {
        return fmt.Sprintf("Welcome %s!", p.Name)
}
```

The method on the struct can be called via dot notation.

```
p := Person{Name: "Bronson"}
fmt.Println(p.Greetings())
// Output: Welcome Bronson!
```

Notice the way we called the method `Greetings()` on the `Person` instance `p`. It's exactly like the way you call methods in an object-oriented programming language.

Remember: a method is just a function with a receiver argument. Methods help to avoid naming conflicts - since a method is tied to a particular receiver type, you can have the same method name on different types.

```
import "math"

type rect struct {
        width, height int
}
func (r rect) area() int {
        return r.width * r.height
}

type circle struct {
        radius int
}
func (c circle) area() float64 {
        return math.Pow(float64(c.radius), 2) * math.Pi
}
```

There are two types of receivers, value receivers, and pointer receivers.

All the methods we have seen so far have a value receiver which means they will receive a copy of the value passed to the method, meaning that any modification done to the receiver inside the method is not visible to the caller.

You can declare methods with pointer receivers in order to modify the value to which the receiver points. This is done by prefixing the type name with a `*`. For example with the `rect` type, a pointer receiver would be declared as `*rect`. Such modifications are visible to the caller of the method as well.

```
type rect struct {
        width, height int
}
func (r *rect) squareIt() {
        r.height = r.width
}

r := rect{width: 10, height: 20}
fmt.Printf("Width: %d, Height: %d\n", r.width, r.height)
// Output: Width: 10, Height: 20

r.squareIt()
fmt.Printf("Width: %d, Height: %d\n", r.width, r.height)
```

```
// Output: Width: 10, Height: 10
```

Instructions

Note: This exercise is a continuation of the `need-for-speed` exercise.

In this exercise you'll be organizing races between various types of remote controlled cars. Each car has its own speed and battery drain characteristics.

Cars start with full (100%) batteries. Each time you drive the car using the remote control, it covers the car's speed in meters and decreases the remaining battery percentage by its battery drain.

If a car's battery is below its battery drain percentage, you can't drive the car anymore.

The remote controlled car has a fancy LED display that shows two bits of information:

- The total distance it has driven, displayed as: `"Driven <METERS> meters"`.
- The remaining battery charge, displayed as: `"Battery at <PERCENTAGE>%"`.

Each race track has its own distance. Cars are tested by checking if they can finish the track without running out of battery.

1. Drive the car

Implement the `Drive` method on the `Car` that updates the number of meters driven based on the car's speed, and reduces the battery according to the battery drainage:

```
speed := 5
batteryDrain := 2
car := NewCar(speed, batteryDrain)
car.Drive()
// car is now Car{speed: 5, batteryDrain: 2, battery: 98, distance: 5}
```

Note: You should not try to drive the car if doing so will cause the car's battery to be below 0.

2. Display the distance driven

Implement a `DisplayDistance` method on `Car` to return the distance as displayed on the LED display as a `string`:

```
speed := 5
batteryDrain := 2
car := NewCar(speed, batteryDrain)

fmt.Println(car.DisplayDistance())
// Output: "Driven 0 meters"
```

3. Display the battery percentage

Implement the `DisplayBattery` method on `Car` to return the battery percentage as displayed on the LED display as a `string`:

```
speed := 5
batteryDrain := 2
car := NewCar(speed, batteryDrain)

fmt.Println(car.DisplayBattery())
// Output: "Battery at 100%"
```

4. Check if a remote control car can finish a race

To finish a race, a car has to be able to drive the race's distance. This means not draining its battery before having crossed the finish line. Implement the `CanFinish` method that takes a `trackDistance int` as its parameter and returns `true` if the car can finish the race; otherwise, return `false`:

```
speed := 5
batteryDrain := 2
car := NewCar(speed, batteryDrain)

trackDistance := 100

car.CanFinish(trackDistance)
// => true
```

Source

Created by

- @tehsphinx

Contributed to by

- @oanaOM
- @mcastorina